| | |
|---|---|
| Team Number : | apmcmlz2300406 |
| Problem Chosen : | |

# 2023 APMCM Wuyue Cup summary sheet

With the rapid development of technology and the increasing demand for computational tasks, the need for a more efficient and effective information infrastructure has become increasingly important. The Computing Power Network (CPN) is a novel solution that aims to meet these demands by strategically allocating and scheduling computing resources based on business needs.

For the first question, we can define whether each location is a server as a quantum bit, and just compute the sum of the server's service areas, with the caveat that the overlapping areas are computed in the same way. In the end it is enough to add the constraint that there are only two servers.

For the second question, which is the construction of servers and the deployment of resources, we can consider where each user arithmetic demand goes and whether that demand is passed to the corresponding server as a quantum bit. For each user arithmetic demand, there are only three choices, corresponding to three cost functions, and these are easy to obtain, they can only be one of them. However, it is tricky to determine whether the server exceeds the capacity limit. We introduce classical parameters to characterize whether the demand received by the server exceeds the capacity cap, so that different cost functions can be selected. Moreover, we use slack variables to satisfy such inequality relations. The final number of quantum bits required is only 73.

For problem 3, we model portfolio optimization with respect to the use of kaiwu.

## Table of contents

# 1 Introduction

## 1.1 Background

**Computing power network (CPN)** consists of **end users**, **edge servers** and **cloud servers**. The function of allocating and scheduling computing resources for end-user demand. And the appropriate deployment of edge servers in the space to allocate computing resources helps to reduce latency, lower costs, and improve overall network efficiency and user experience.

When using traditional computers and algorithms for coordinated planning of edge server siting and computational resource allocation, the computational complexity of accurately solving combinatorial optimization problems grows exponentially as the scale of the computational tasks continues to grow, which we call np (Nondeterministic polynominal) problems, and the computational time and resources required are also very expensive. The computational time and resources required are also very expensive. However, quantum computing has become a promising technology with significant advantages over traditional computers for certain operations optimization problems. The Coherent Ising Machine (CIM), as a specialized quantum computing device, also provides new ideas for solving the CPN deployment problem.

## 1.2 Question Restatement

**Problem 1:** Depending on the distribution of computational requirements, two edge servers are deployed in the grid area with a coverage radius of 1. The QUBO model is built in this way. The deployment coordinates of the edge servers that **provide the maximum coverage of the computational demand**, and the corresponding total computational demand, are solved using the simulated annealing solver and the CIM simulator of the Kaiwu SDK.

**Problem 2:** End-users incur costs when connecting to cloud servers or edge servers. Based on the computational requirements such as user geographic location, edge server candidate coordinates and cost, a QUBO model with as few bits as possible is built and the optimal computational network layout is solved using the simulated annealing solver of the Open Matter SDK and the CIM simulator of the SDK.

**Problem 3:** Propose a potential decision optimization application scenario which builds a suitable QUBO model, to **show that the scenario should have practical value, scalability, real business needs, and demonstrate the benefits of Coherent Ising Machine** (CIM).

## 2 Symbol Description

| Symbol | Explain |
|---|---|
| $a_{i,j}$ | the amount of computation requested by the user at position $(i,j)$. |
| $x_{i,j}$ | qubits to be solved, $x_{i,j} = \begin{cases} 1, & \text{An edge server exists at location } (i,j). \\ 0, & \text{No edge server exists at location } (i,j). \end{cases}$ |
| $x^c_{i,j}$ | qubits to be solved, $x^c_{i,j} = \begin{cases} 1, & \text{The user } (i,j) \text{at positionis connected to the cloud server.} \\ 0, & \text{The user } (i,j) \text{ is not connected to the cloud server.} \end{cases}$ |
| $x^r_{i,j}$ | qubits to be solved, $x^r_{i,j} = \begin{cases} 1, & \text{The user } (i,j) \text{ is connected to the edge server } \boldsymbol{r}. \\ 0, & \text{The user } (i,j) \text{ is not connected to the edge server } \boldsymbol{r}. \end{cases}$ |
| $d^C_{i,j}$ | the data transmission distance from the user$(i,j)$ to the cloud service. |
| $F_r$ | fixed cost required to build an edge server $\boldsymbol{r}$. |
| $\Delta^r$ | the amount of computation that is forwarded to the cloud service for computation via the edge server. |
| $S_r$ | qubits to be solved, $S_r = \begin{cases} 1, & \text{Building edge servers } \boldsymbol{r} \\ 0, & \text{not build edge servers } \boldsymbol{r} \end{cases}$ |
| $y^r$ | classical parameter, $y^r = \begin{cases} 1, & \text{Edge servers } r \text{ can do compute on their own} \\ 0, & \text{Edge servers } r \text{ can't do compute on their own} \end{cases}$ |
| $H^C$ | cost Function functions using cloud computing. |
| $H^r_{i,j}$ | denotes the overhead function of the user at location $(i,j)$ using the edge calculator $\boldsymbol{r}$. |
| $k_T$ | ransmission cost per unit distance for unit arithmetic demand propagation to the corresponding server |
| $k_C$: | computational cost per unit of arithmetic demand at the corresponding server |

## 3 Problem Analysis and Solution for Question 1

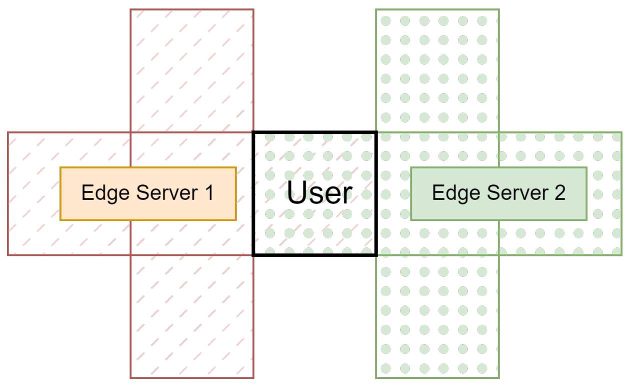
### 3.1 Analysis for Question 1

In order to be able to use the QUBO model to find the edge server location that covers the most computations, we need to determine the sum of computations associated with the selection in case of location (i,j), which we call the objective function . In this way, we built the QUBO model and solved it using the simulated annealing solver and the CIM simulator of the Kaiwu SDK.

### 3.2 Model Esatblishment: Determine the Objective Function

For each user, his/her own computational demand is satisfied if and only if there is an edge server in his/her "neighborhood" ($x_{i,j}=1$), but even if there is more than one server, his/her own demand will be satisfied only once. Therefore, the value of the demand that one contributes is the value of one's own computed demand multiplied by the bitwise logical **"OR"** value of $x_{i,j}$ of the 5 points of the "neighbor". The total satisfied demand is the sum of the demand values contributed by all users.

**Figure 1.** Situation when two edge servers overlap and cover some users at the same time.



Taking the user $(i,j)$ as the center and the coverage radius of the edge server as the radius, traverse all the surrounding nodes for the presence of edge servers, and for each edge server present, the total coverage is increased by $a_{i,j}\sum_{m,n}^{R_r} x_{m,n}$ . If there are two edge servers in the range, you need to subtract the double-counted portion, i.e., $\frac{1}{2}\sum_{i,j} a_{i,j}\sum_{m,n}^{R_r} x_{m,n}\sum_{R_r} x_{p,q}$.

Thus, we can sum up the cases for all users. The number of demands that can be satisfied by a distribution $\{\mathrm{x_{i,j}}\}$ is:

$$D = \sum_{i,j} a_{i,j} \sum_{m,n}^{R_r} x_{m,n}\left(1 - \frac{1}{2}\sum_{R_r} x_{p,q}\right) \tag{1}$$

Where $R_r = |m-i| \leqslant 1, |n-j| \leqslant 1$, and $x_{i,j} = \begin{cases} 1, & \text{An edge server exists at location } (i,j). \\ 0, & \text{No edge server exists at location } (i,j). \end{cases}$

The constraint to be satisfied is that the number of edge servers is 2, i.e.

$$\sum_{i,j} x_{i,j} = 2 \tag{2}$$

The objective function can be expressed as:

$$H = H_A + H_B \tag{3}$$

where $H_A$ is the constraint QUBO form and $H_B$ is the opposite of the demand that can be satisfied. Noting that $x{=}x^2$ (bits take only 0 or 1), there are

$$H_A = A\left(2 - \sum_{i,j} x_{i,j}\right)^2 \tag{4}$$

$$H_B = -D = -\sum_{i,j} a_{i,j}\left(\sum_{m,n}^{R_r} x_{m,n}^2 - \frac{1}{2}\sum_{m,n}^{R_r}\sum_{p,q}^{R_r} x_{m,n}x_{p,q}\right) \tag{5}$$

$(\sum_{i,j} f_{i,j}(x_{i,j}) = \sum_{n=1}^{16} f_{4(i-1)+j}(x_n))$

In the QUBO model, we constrain our constraints by adding weights before the value A as penalty factor. If the penalty factor $A$ is large enough, in order to ensure that the objective function H is minimized, it must be such that $H_B$ is minimized (demand $D$ is maximized) while satisfying $H_A = 0$.

## 3.3 Problem Solving: Code implementation

### 3.3.1 Find the quadratic coefficient matrix $B$ of $H_B$

Create the $16 \times 16$ empty matrix $B$. Traverse the demand matrix **Demand**, and for each position $(i,j)$, add $-a_{i,j}$ to $B$ at the position corresponding to $x_{i,j}$ and its "neighbors"; then add $a_{i,j}$ to the non-diagonal elements of these 5 bits (**10 per position, no repeatedly taken**).

### 3.3.2 Establishing the objective function

Use the **`kw.qubo.ndarray()`** method in the **`kaiwu`** library to get the bitvector $x = x_n, n = 1, 2, \ldots 16$ (qubo variable), and then implement the summation notation with **`kw.qubo.sum()`**, following the following logic to implement the code ($B_{a,b}$ is the matrix element of $B$ in $(a,b)$) :

$$\textbf{\textit{servers\_num\_limit}} = H_A = A\left(2 - \sum_{n=1}^{16} x_n\right)^2$$

$$\textbf{\textit{negative\_demand}} = H_B = \sum_{a=1}^{16}\sum_{b=1}^{16} x_a x_b B_{a,b}$$

The objective function: $\textbf{\textit{obj}} = H = H_A + H_B$

Deliver the objective function ***obj*** to the quantum simulated annealer to obtain the value of the bit vector $x$. Verify that $H_A = 0$ is valid (if not, increase the penalty factor $A$ and solve again). After that, according to the bit subscript conversion rule, the index of the non-zero value of $x$ is converted to two-dimensional coordinates, and the location map matrix of the edge server is obtained.

**(see annex for specific code)**

## 3.4 The Results of Problem 1

**Table 1-1. Demand Coverage Table**

| Total Computational Demand Covered |
|---|
| 419 |

**Table 1-2. Edge Activation Status Table**

| Number | Edge Coordinates | Covered Grid Coordinates | Covered Computational Demand |
|---|---|---|---|
| 1 | [3,1] | [3,1][2,1][3,1][4,1][3,2] | 187 |
| 2 | [2,3] | [2,3][2,2][1,3][3,3][2,4] | 232 |

# 4 Analysis and Solution for Question 2

## 4.1 Analysis for Question 2

End-users can choose to connect to a cloud server or an edge server and fulfill all their requirements in one server. When an edge server's compute demand exceeds its capacity limit, the excess demand is assigned to a cloud server. Each edge server has a capacity of 12 available compute resources, while the cloud server has no compute resource capacity limit. The coverage radius of the edge servers is 3, while the coverage radius of the cloud servers is unlimited. In summary, in order to optimize the best edge server deployment scenario using the QUBO model, we need to discuss the individual scenarios in a categorical manner.

## 4.2 Model Esatblishment: Determine the Objective Function

Upon analysis, it can be seen that the cost function has three components: the computation and data transmission cost of using the edge server, the computation and data transmission cost of using the cloud server, and the computation and transmission cost due to the portion of the edge server that receives requests for more than 12 units of computational resources that are forwarded to the cloud server via the edge server.

According to the relevant content within the annex, we can set $a_{i,j}$ as the computational resource requirement for end user $(i,j)$. Use quantum bit $x_{i,j}^{C}$ as the judgment of whether the user $(i,j)$ chooses to link to the cloud service or not. The cost function $H^{C}$ for the using cloud computing is:

$$H^c = k_T^c \times \sum_{i,j} a_{i,j} d_{i,j}^c x_{i,j}^c + k_C^c \times \sum_{i,j} a_{i,j} x_{i,j}^c \tag{6}$$

Use quantum bit $x_{i,j}^r$ as a judgment of whether user $(i,j)$ chooses to link to edge server $r$.Denote the data transmission distance from the user to the cloud service by $d_{i,j}^{C}$. Denote the data transmission distance from the user to the edge server $r$ by $d_{i,j}^r$. Denotes that the cost function about the edge servers is :

$$H_1^r = k_T^e \times \sum_{i,j}^{R_r} a_{i,j} d_{i,j}^r x_{i,j}^r + k_C^e \times \sum_{i,j}^{R_r} a_{i,j} x_{i,j}^r + S_r F_r \tag{7}$$

where $S_r$ is the qubit used to refer to whether or not to build the edge server $r$. $F_r$ is the fixed overhead required to build the edge server, $r$.

The above equation does not take into account the case where the edge server receives more requests for computing resources than it can handle, and we will discuss this in more detail in the following constraint relation.

When an edge server receives a request for computational resources that exceeds its capacity, we need to compensate for the excess, so that the excess is corrected from the "cost of incorrectly allocating computational resources to the edge server" to the cost that should have been processed by the edge server and handed over to the cloud server. We define a set of arithmetic difference functions $\sum_{i,j}^{R_r} x_{i,j}^r a_{i,j} - 12$, where 12 is the upper capacity limit. The compensation function is :

$$C^r = -k_C^e \Delta^r + k_C^c \Delta^r + k_T^{ec} \Delta^r d_{r-c}^r$$

In the event that an edge server receives arithmetic requests that exceed the capacity limit, the cost function for all arithmetic requests received by that $r$th edge server is:

$$H_2^r = H_1^r + C^r \tag{8}$$

Considering that the capacity limit is binary, we can introduce a set of parameters $y^r$ characterizing whether the arithmetic requests accepted by this server exceed the upper limit, $y^r = \left\{ \begin{array}{ll} 1, & \text{Edge servers } r \text{ can do compute on their own} \\ 0, & \text{Edge servers } r \text{ can}'t \text{ do compute on their own} \end{array} \right\}$

We want $y^r = 1$ for the compensation function to take effect, hence writing as $y_r C^r$.

Also $y^r = 1$ , the necessary condition is $\Delta^r \geqslant 0$, i.e.

$$y^r \Delta^r \geqslant 0 \tag{9}$$

There are some constraints between several quantum bits:

1. Each location $(x,y)$ must use one of all the servers including cloud services, the corresponding constraint equation is :

$$G_1^{i,j} = \left( \sum_r x_{i,j}^r - 1 \right)^2 = 0 \tag{10}$$

2. If the edge server r is not established, i.e., $S_r = 0$, then each user (i, j) cannot connect to the edge server $r$. The constraint equation for this is :

$$G_2^r = (1 - S_r) \sum_{i,j}^{\text{range}} x_{i,j}^r = 0 \tag{11}$$

3. Define $h^r$ to be a nonnegative slack variable. There must be a G_3 > 0 (penalty) when Δ_r < 0, and Δ_r ⩾ 0 can be regulated by h^r to make G_3 = 0, thus incentivizing it to be necessarily overdetermined at y^r = 1, thus circumventing the erroneous negative C result.

So here we can introduce the slack variable $h^r$. When setting $y^r = 1$, if $\Delta^r \geqslant 0$, we can adjust $h^r$ to make constraint 3 is 0. If $\Delta^r \geqslant 0$, the slack variable can not reach the adjustment. Similarly we can do the same form of constraint for the case $y^r = 0$, except that $\Delta^r \geqslant 0$. The final Hamiltonian therefore takes the following form:

$$H_{\text{cost}} = H^c + \sum_r (H_1^r + y^r C^r) + \sum_{i,j} \lambda_1^{i,j} G_1^{i,j} + \sum_r \lambda_2^r G_2^r + \sum_r \lambda_3^r y^r G_3 + \sum_r \lambda_4^r (1 - y^r) G_4$$

where $\lambda$ is the penalty factor, and $y_r$ is the classical parameter, so the Hamiltonian is still quadratic.

## 4.3 The Results of Problem2

**Table 2-1.** Total Cost Table

| Total cost | | | | |
|---|---|---|---|---|
| 488.42 | | | | |
| Fixed Cost | Computational Cost | Transmission Cost | | |
| 172 | 47 | 269.42 | | |
| | | User-Edge | Edge-Cloud | User-Cloud |
| | | | | |

**Table 2-2.** Edge Activation Status Table

| Number | Edge Coordinates | Activation States |
|---|---|---|
| 1 | (6,1) | 1 |
| 2 | (2,3) | 1 |
| 3 | (4,5) | 1 |
| 4 | (6,5) | 1 |
| 5 | (2,1) | 1 |

**Note**: The 'Activation Status' field in the table: 1 represents the establishment of an edge server at that location, and 0 represents non-activation.

**Table 2-3.** User-to-Edge Transmission of Computational Demand Table

| Number | Edge Coordinates | User Coordinates | Amount of Computational demand fortransmission |
|---|---|---|---|
| 1 | (6,1) | | |
| 2 | (2,3) | (1, 3), (3, 3), (4, 1), (5, 3) | |
| 3 | (4,5) | (4, 4),(5, 5) | |
| 4 | (6,5) | (6, 6) | |
| 5 | (2,1) | (2, 2) | |

**Table 2-4.** User-to-Cloud Transmission of Computational Demand Table

| Number | Cloud Coordinates | User Coordinates | Amount of Computational demand for transmission |
|---|---|---|---|
| 1 | (4, 0) | (2, 4),(2, 5),(3, 6),(6, 2),(6, 6) | |

# Appendix A. The code for Problem 1

```
# 2023 APMCM Wuyue Cup - Problem 1
import numpy as np
import kaiwu as kw

# the relative coordinates of neighbors (including themselves)
dx = [0, -1, 1, 0, 0]
dy = [0, 0, 0, -1, 1]
# number of bits
n = 16
# penalty for incorrect number of servers
L = 10 ** 3


# Marks conversion for bits
def n_(i, j):
    return 4 * i + j


def p_(z):
    return [int(z / 4), z % 4]


# Add v to the position(n(i1,j1),n(i2,j2)) of input H
def Hv(H, i1, j1, i2, j2, v):
    if 0 <= i1 < 4 and 0 <= j1 < 4 and 0 <= i2 < 4 and 0 <= j2 < 4:
        H[n_(i1, j1)][n_(i2, j2)] += v


# demand map
A = np.array([[36, 51, 54, 48],
              [20, 46, 34, 35],
              [63, 74,  5, 46],
              [54, 44, 27, 38]])

# 'negative demand satisfaction matrix', B
MB = [[0 for j in range(n)] for i in range(n)]

for i in range(4):
    for j in range(4):
        d = A[i][j]
        # diagonal element
        for k in range(5):
            Hv(MB, i + dx[k], j + dy[k], i + dx[k], j + dy[k], -d)
        # non-diagonal element
        for a in range(0, 5):
            for b in range(a+1, 5):
                Hv(MB, i + dx[a], j + dy[a], i + dx[b], j + dy[b], d)
```

```
# Create qubo variable
x = kw.qubo.ndarray(n, ‘‘x", kw.qubo.binary)


# Server limit penalty function
HA = L * (2 - kw.qubo.sum(x[a] for a in range(n))) ** 2
# Negative demand function
HB = kw.qubo.sum(kw.qubo.sum(x[a] * x[b] * MB[a][b] for b in range(n)) for a in range(n))
# Objective function (H)
obj = HA + HB


# Parse QUBO
obj = kw.qubo.make(obj)
# Convert to Ising model
obj_ising = kw.qubo.cim_ising_model(obj)
# Extract the Ising matrix
matrix = obj_ising.get_ising()["ising"]


# Perform calculation using CIM simulator
output = kw.cim.simulator(
                matrix,
                pump = 1.3,
                noise = 0.2,
                laps = 5000,
                dt = 0.05,
                normalization = 0.3,
                iterations = 50)

# Sort the results
opt = kw.sampler.optimal_sampler(matrix, output, bias = 0, negtail_ff = False)
# Select the best solution
cim_best = opt[0][0]
# If the linear term variable is -1, perform a flip
cim_best = cim_best * cim_best[-1]

# Print the spin value
print("spin: {}".format(cim_best))


# Get the list of variable names
var_s = obj_ising.get_variables()
# Substitute the spin vector and obtain the result dictionary
sol_dict = kw.qubo.get_sol_dict(cim_best, var_s)

snl_val = kw.qubo.get_val(HA, sol_dict)
print('servers_num_limit(HA): {}'.format(snl_val))

negative_demand_val = kw.qubo.get_val(HB, sol_dict)
print('negative_demand(-HB): {}'.format(negative_demand_val))


if snl_val != 0:
    print('\nInvalid path.')
    print('\nPlease try again.')
else:
    print('\nValid path.')
    print('Satisfied demand: ', end='')
    print(int(-negative_demand_val))

    # Get the numerical value matrix of x
    x_val = kw.qubo.get_array_val(x, sol_dict)
    # Find the indices of non-zero items
    nonzero_index = np.array(np.nonzero(x_val)).T
    orders = nonzero_index[:].flatten()

    # Print the path order
    print('The coordinates of the edge servers:')
    print('Number | (X-axis, Y-axis)')

    for o in range(len(orders)):
        co = p_(orders[o])
        print(o+1, end=' | (')
        print(co[1]+1, end=', ')
        print(co[0]+1, end=')\n')
```

## Appendix B. The code for Problem 2

```
# 2023 APMCM Wuyue Cup - Problem 2
import numpy as np
import kaiwu as kw


def euclidean_distance(i1, j1, i2, j2):
    return round(((i1 - i2) ** 2 + (j1 - j2) ** 2) ** 0.5, 2)


# determines whether (i, j) is covered by the scope of edge server r
def isCoveredBy(r, i, j):
    return euclidean_distance(Ei[r], Ej[r], i, j) <= 3


# return 1 if (i, j) is covered by the scope of r, else 0
def R(r, i, j):
    return 1 if isCoveredBy(r, i, j) else 0


# penalty coefficients
L1 = 10 ** 9
L2 = 10 ** 8
L3 = 10 ** 3
L4 = 10 ** 4
L5 = 10 ** 8

# transmission cost and calculation cost
kT_C = 2
kT_E = 1
kT_CE = 1
kC_C = 1
kC_E = 2

# edge server capacity
Cap = 12

# cloud server's coordinate
Cc = [4, 0]
Ci = Cc[1] - 1
Cj = Cc[0] - 1

# edge servers' coordinates and fixed cost
F = [70, 52, 56, 64, 40]
Ex = [6, 2, 4, 6, 2]
Ey = [1, 3, 5, 5, 1]
Ei = [0 for i in range(5)]
Ej = [0 for j in range(5)]
for i in range(5):
    Ei[i] = Ey[i] - 1
for j in range(5):
    Ej[j] = Ex[j] - 1

# distance between users and cloud server
Dc = [[0 for j in range(6)] for i in range(6)]
# distance between users and edge servers
D = [[[0 for j in range(6)] for i in range(6)] for r in range(5)]
# distance between edge servers and cloud server
Dce = [0 for r in range(5)]
for i in range(6):
    for j in range(6):
        Dc[i][j] = euclidean_distance(i, j, Ci, Cj)
        for r in range(5):
            D[r][i][j] = euclidean_distance(i, j, Ei[r], Ej[r])
            Dce[r] = euclidean_distance(Ei[r], Ej[r], Ci, Cj)

# demand map
A = np.array([[0, 0, 3, 0, 0, 0],
              [0, 0, 0, 4, 6, 0],
              [0, 0, 4, 0, 0, 7],
              [4, 0, 0, 11, 0, 0],
              [0, 0, 8, 0, 3, 0],
              [0, 5, 0, 0, 0, 5]])

# the number of bits which are used for indicate connectivity to the servers, x_ijr, count from 0
NB = -1
NBc = -1
# the distribution network of bits x_ijc
Bc = [[-1 for j in range(6)] for i in range(6)]
for i in range(6):
    for j in range(6):
        if A[i][j] > 0:
            NBc += 1
            NB += 1
            Bc[i][j] = NB

# the distribution network of bits x_ijr
Br = [[[-1 for j in range(6)] for i in range(6)] for r in range(5)]
for r in range(5):
    for i in range(6):
        for j in range(6):
            if A[i][j] > 0:
                if isCoveredBy(r, i, j):
                    NB += 1
                    Br[r][i][j] = NB


# return the coordinates of the n-th bit in the distribution network
def coordinateOfBit(z):
    for i in range(6):
        for j in range(6):
            if Bc[i][j] == z:
                return [i, j]
    for i in range(6):
        for j in range(6):
            for r in range(5):
                if Br[r][i][j] == z:
                    return [r, i, j]


# from now on NB count from 1
NB += 1
NBc += 1


def printC(n, i, j):
    print(n + 1, end=' | (')
    print(i + 1, end=', ')
    print(j + 1, end=')\n')


def count(array, v):
    c = 0
    for i in range(len(array)):
        if v == array[i]:
            c += 1
    return c


# Semi-classical iteration, correcting y based on the result of delta_r at each round
def IterationY(t, y, final):
    OB = count(y, 1) * 6
    od = 0
    ods = [-1 for _ in range(5)]
    for rc in range(5):
        if y[rc] == 1:
            ods[rc] = od
            od += 1

    # Create qubo variable
    x = kw.qubo.ndarray(NB + 5 , ``x", kw.qubo.binary)

    # return 1 if (i, j) has demand(a_ij > 0), else 0
    def M(i, j):
        return 1 if A[i][j] > 0 else 0

    # bits that indicate the connectivity to the cloud servers, select non-zero bits to be summed here
    def Xc(i, j):
        return x[0] * 0 if Bc[i][j] < 0 else x[Bc[i][j]]
    # bits that indicate the connectivity to the edge servers, the principle is same as above
    def Xr(r, i, j):
        return x[0] * 0 if Br[r][i][j] < 0 else x[Br[r][i][j]]

    # bits that indicate whether to build the edge server r
    def Sr(r):
        return x[NB + r]

    # the difference between capacity limit and the total demand among the range of edge server r
    def delta_r(r):
        return kw.qubo.sum(kw.qubo.sum(A[i][j] * Xr(r, i, j) * R(r, i, j) for j in range(6)) for i in
range(6)) - Cap

    # Classical bits, updated in iterations, indicate whether edge server r needs to connect to the cloud
    # Y = [0 for r in range(5)]

    def Hr(r):
        if y[r] == 0:
            return x[0] * 0
        else:
            return kw.qubo.sum((2 ** p) * x[NB + 5 + (6 * ods[r]) + p] for p in range(6))

    # cost of cloud server
    Hc = kw.qubo.sum(kw.qubo.sum((Dc[i][j] * kT_C + kC_C) * A[i][j] * Xc(i, j) for j in range(6)) for i in
range(6))

    # cost of all edge server
    He = kw.qubo.sum(
        (kw.qubo.sum(kw.qubo.sum((D[r][i][j] * kT_E + kC_E) * A[i][j] * Xr(r, i, j) * R(r, i, j) for j in
range(6))
                     for i in range(6)) + Sr(r) * F[r]) for r in range(5))

    # compensation cost
    C = kw.qubo.sum((Dce[r] * kT_CE + kC_C - kC_E) * (
            kw.qubo.sum(kw.qubo.sum(A[i][j] * Xr(r, i, j) * R(r, i, j) for j in range(6)) for i in range(6))
- Cap) *
                    y[r] for r in range(5))

    # Constraint 1: Each location can only use one of all servers, including the cloud
    G1 = L1 * kw.qubo.sum(
        kw.qubo.sum(((Xc(i, j) + kw.qubo.sum(Xr(r, i, j) for r in range(5)) - 1) ** 2) * M(i, j) for j in
range(6)) for
        i in range(6))

    # Constraint 2: Edge servers can only be used after they are constructed
    G2 = L2 * kw.qubo.sum(
        kw.qubo.sum(kw.qubo.sum(Xr(r, i, j) * R(r, i, j) for j in range(6)) for i in range(6)) * (1 - Sr(r))
        for r in range(5))

    G3 = L3 * kw.qubo.sum(((Hr(r) - delta_r(r)) ** 2) * y[r] for r in range(5))
    G4 = L4 * kw.qubo.sum(((Hr(r) + delta_r(r)) ** 2) * (1 - y[r]) for r in range(5))
    G5 = L5 * kw.qubo.sum(y[r] * (1 - Sr(r)) for r in range(5))

    # Total Cost
    HT = Hc + He + C
    # Objective function (H)
    obj = HT + G1 + G2 + G3 + G4 + G5

    # Parse QUBO
    obj = kw.qubo.make(obj)
    # Convert to Ising model




    obj_ising = kw.qubo.cim_ising_model(obj)
    # Extract the Ising matrix
    matrix = obj_ising.get_ising()["ising"]

    # Perform calculation using CIM simulator
    output = kw.cim.simulator(
        matrix,
        pump=1.3,
        noise=0.2,
        laps=5000,
        dt=0.05,
        normalization=0.3,
        iterations=50)

    # Sort the results
    opt = kw.sampler.optimal_sampler(matrix, output, bias=0, negtail_ff=False)
    # Select the best solution
    cim_best = opt[0][0]
    # If the linear term variable is -1, perform a flip
    cim_best = cim_best * cim_best[-1]

    # Get the list of variable names
    var_s = obj_ising.get_variables()
    # Substitute the spin vector and obtain the result dictionary
    sol_dict = kw.qubo.get_sol_dict(cim_best, var_s)
    g1_val = kw.qubo.get_val(G1, sol_dict)
    g2_val = kw.qubo.get_val(G2, sol_dict)
    # g3_val = kw.qubo.get_val(G3, sol_dict)
    # g4_val = kw.qubo.get_val(G4, sol_dict)
    g5_val = kw.qubo.get_val(G5, sol_dict)
    c_val = kw.qubo.get_val(C, sol_dict)
    cost_val = kw.qubo.get_val(HT, sol_dict)

    if g1_val != 0 or g2_val != 0 or g5_val != 0:
        print('x')
        return False

    print('{}'.format(t), end=' ')
    print('[{}'.format(g1_val), end=' | ')
    print('{}'.format(g2_val), end=' | ')
    # print('{}'.format(g3_val), end=' | ')
    # print('{}'.format(g4_val), end=' | ')
    print('{}]'.format(g5_val), end=' ')
    print('{}'.format(Y), end=' ')

    # Get the numerical value matrix of x
    x_val = kw.qubo.get_array_val(x, sol_dict)
    # Find the indices of non-zero items
    nonzero_index = np.array(np.nonzero(x_val)).T
    orders = nonzero_index[:].flatten()

    Srs = [0 for _ in range(5)]
    for o in range(len(orders)):
        n = orders[o]
        if NB <= n < NB + 5:
            Srs[n - NB] = 1
    print('{}'.format(Srs), end=' ')
    print('{}'.format(round(c_val), 2), end=' ')
    print('{}'.format(round(cost_val), 2), end=' [')

    for r in range(5):
        dr = kw.qubo.get_val(delta_r(r), sol_dict)
        print('{}'.format(dr), end=(']\n' if r == 4 else ', '))

    if not final:
        return True

    Cis = 0
    UC = [[0 for _ in range(6)] for _ in range(6)]
    UE = [[[0 for _ in range(6)] for _ in range(6)] for _ in range(5)]
    for o in range(len(orders)):
        n = orders[o]
        if n < NB:
            co = coordinateOfBit(n)
            if n < NBc:
                UC[co[0]][co[1]] = 1
                Cis += kC_C * A[co[0]][co[1]]
            else:
                UE[co[0]][co[1]][co[2]] = 1
                Cis += kC_E * A[co[0]][co[1]]
        else:
            if NB <= n < NB + 5:
                Srs[n - NB] = 1

    print('Users who connect directly to the cloud server:')
    for i in range(6):
        print(UC[i])
    for r in range(5):
        print('Users who connect to the edge server r{}:'.format(r + 1))
        for i in range(6):
                print(UE[r][i])
        print()

        Con = 0
        for r in range(5):
            Con += Srs[r] * F[r]

        Tis = cost_val - Con - Cis

        print('-Final Result-')
        print('Total Cost: {}'.format(round(cost_val, 2)))
        print('Construction Cost: {}'.format(round(Con, 2)))
        print('Transmission Cost: {}'.format(round(Tis, 2)))
        print('Calculation Cost: {}'.format(round(Cis, 2)))

        print('The coordinates of edge servers constructed:')
        for r in range(5):
            print('', end='(')
            print(Ex[r], end=', ')
            print(Ey[r], end=')\n')

        print('The coordinates of the users who connect directly to the cloud server:')
        print('Number | (X-axis, Y-axis)')
        n = 0
        for i in range(6):
            for j in range(6):
                if UC[i][j] == 1:
                    printC(n, i, j)
                    n += 1
        print('The coordinates of the users who connect directly to the edge server:')
        for r in range(5):
            print('Edge server {}, or '.format(r+1), end='(')
            print(Ex[r], end=', ')
            print(Ey[r], end=')\n')
            print('Number | (X-axis, Y-axis)')
            n = 0
            for i in range(6):
                for j in range(6):
                    if UE[r][i][j] == 1:
                        printC(n, i, j)
                        n += 1

        return True


    T = 0
    print('T [ G1 | G2  |  G3  |  G4 | G5 ]        Yr            Sr        C        Cost                delta_r')
    for y1 in range(2):
        for y2 in range(2):
            for y3 in range(2):
                for y4 in range(2):
                    T += 1
                    Y = [y1, y2, y3, y4, 0]
                    while True:
                        if IterationY(T, Y, False):
                            break


    T = 0
    Y = [0, 0, 1, 1, 0]
    while True:
        if IterationY(T, Y, True):
            break
```